2

VISUALIZING DATA WITH GRAPHS

In this chapter, you'll learn a powerful way to present numerical data: by drawing graphs with Python. We'll start by discussing the number line and the Cartesian plane. Next, we'll learn about the powerful plotting library *matplotlib* and how we can use it to create graphs. We'll then explore how to make graphs that present data clearly and intuitively. Finally, we'll use graphs to explore Newton's law of universal gravitation and projectile motion. Let's get started!

Understanding the Cartesian Coordinate Plane

Consider a *number line*, like the one shown in Figure 2-1. Integers from –3 to 3 are marked on the line, but between any of these two numbers (say, 1 and 2) lie all possible numbers in between: 1.1, 1.2, 1.3, and so on.

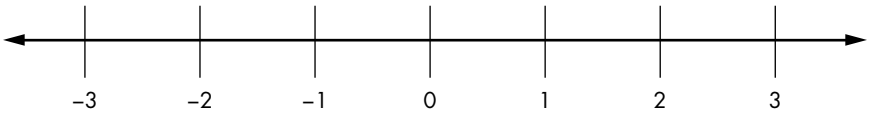

Figure 2-1: A number line

The number line makes certain properties visually intuitive. For example, all numbers on the right side of 0 are positive, and those on the left side are negative. When a number a lies on the right side of another number b, a is always greater than b and b is always less than a.

The arrows at the ends of the number line indicate that the line extends infinitely, and any point on this line corresponds to some real number, however large it may be. A single number is sufficient to describe a point on the number line.

Now consider two number lines arranged as shown in Figure 2-2. The number lines intersect at right angles to each other and cross at the 0 point of each line. This forms a *Cartesian coordinate plane*, or an x-y plane, with the horizontal number line called the x-axis and the vertical line called the y-axis.

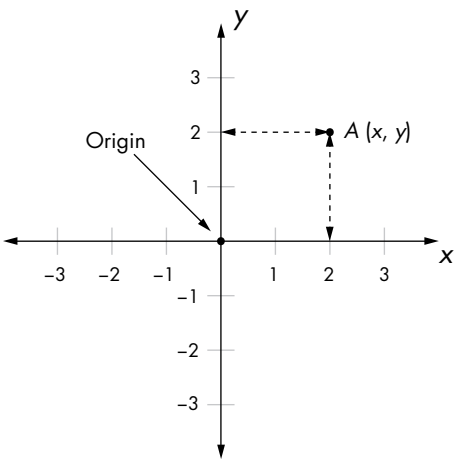

Figure 2-2: The Cartesian coordinate plane

As with the number line, we can have infinitely many points on the plane. We describe a point with a pair of numbers instead of one number. For example, we describe the point A in the figure with two numbers, x and y, usually written as (x, y) and referred to as the *coordinates* of the point.

As shown in Figure 2-2, *x* is the distance of the point from the origin along the *x*-axis, and *y* is the distance along the *y*-axis. The point where the two axes intersect is called the *origin* and has the coordinates (0, 0).

The Cartesian coordinate plane allows us to visualize the relationship between two sets of numbers. Here, I use the term *set* loosely to mean a collection of numbers. (We'll learn about mathematical sets and how to work with them in Python in Chapter 5.) No matter what the two sets of numbers represent—temperature, baseball scores, or class test scores—all you need are the numbers themselves. Then, you can plot them—either on graph paper or on your computer with a program written in Python. For the rest of this book, I'll use the term *plot* as a verb to describe the act of plotting two sets of numbers and the term *graph* to describe the result—a line, curve, or simply a set of points on the Cartesian plane.

Working with Lists and Tuples

As we make graphs with Python, we'll work with *lists* and *tuples*. In Python, these are two different ways to store groups of values. Tuples and lists are very similar for the most part, with one major difference: after you create a list, it's possible to add values to it and to change the order of the values. The values in a tuple, on the other hand, are immediately fixed and can't be changed. We'll use lists to store *x*- and *y*-coordinates for the points we want to plot. Tuples will come up in "Customizing Graphs" on page 41 when we learn to customize the range of our graphs. First, let's go over some features of lists.

You can create a list by entering values, separated by commas, between square brackets. The following statement creates a list and uses the label `simplelist` to refer to it:

```
>>> simplelist = [1, 2, 3]
```

Now you can refer to the individual numbers—1, 2, and 3—using the label and the position of the number in the list, which is called the *index*. So `simplelist[0]` refers to the first number, `simplelist[1]` refers to the second number, and `simplelist[2]` refers to the third number:

```
>>> simplelist[0]
1
>>> simplelist[1]
2
>>> simplelist[2]
3
```

Notice that the first item of the list is at index 0, the second item is at index 1, and so on—that is, the positions in the list start counting from 0, not 1.

Lists can store strings, too:

```
>>> stringlist = ['a string','b string','c string']
>>> stringlist[0]
'a string'
>>> stringlist[1]
'b string'
>>> stringlist[2]
'c string'
```

One advantage of creating a list is that you don't have to create a separate label for each value; you just create a label for the list and use the index position to refer to each item. Also, you can add to the list whenever you need to store new values, so a list is the best choice for storing data if you don't know beforehand how many numbers or strings you may need to store.

An *empty list* is just that—a list with no items or elements—and it can be created like this:

```
>>> emptylist = []
```

Empty lists are mainly useful when you don't know any of the items that will be in your list beforehand but plan to fill in values during the execution of a program. In that case, you can create an empty list and then use the `append()` method to add items later:

```
❶ >>> emptylist
  []
❷ >>> emptylist.append(1)
  >>> emptylist
  [1]
❸ >>> emptylist.append(2)
  >>> emptylist
❹ [1, 2]
```

At ❶, `emptylist` starts off empty. Next, we append the number 1 to the list at ❷ and then append 2 at ❸. By line ❹, the list is now [1, 2]. Note that when you use `.append()`, the value gets added to the end of the list. This is just one way of adding values to a list. There are others, but we won't need them for this chapter.

Creating a tuple is similar to creating a list, but instead of square brackets, you use parentheses:

```
>>> simpletuple = (1, 2, 3)
```

You can refer to an individual number in `simpletuple` using the corresponding index in brackets, just as with lists:

```
>>> simpletuple[0]
1
```

```
>>> simpletuple[1]
2
>>> simpletuple[2]
3
```

You can also use *negative indices* with both lists and tuples. For example, `simplelist[-1]` and `simpletuple[-1]` would refer to the last element of the list or the tuple, `simplelist[-2]` and `simpletuple[-2]` would refer to the second-to-last element, and so on.

Tuples, like lists, can have strings as values, and you can create an *empty tuple* with no elements as `emptytuple=()`. However, there's no `append()` method to add a new value to an existing tuple, so you can't add values to an empty tuple. Once you create a tuple, the contents of the tuple can't be changed.

Iterating over a List or Tuple

We can go over a list or tuple using a `for` loop as follows:

```
>>> l = [1, 2, 3]
>>> for item in l:
        print(item)
```

This will print the items in the list:

```
1
2
3
```

The items in a tuple can be retrieved in the same way.

Sometimes you might need to know the position or the index of an item in a list or tuple. You can use the `enumerate()` function to iterate over all the items of a list and return the index of an item as well as the item itself. We use the labels `index` and `item` to refer to them:

```
>>> l = [1, 2, 3]
>>> for index, item in enumerate(l):
        print(index, item)
```

This will produce the following output:

```
0 1
1 2
2 3
```

This also works for tuples.

Creating Graphs with Matplotlib

We'll be using matplotlib to make graphs with Python. Matplotlib is a Python *package*, which means that it's a collection of modules with related functionality. In this case, the modules are useful for plotting numbers and making graphs. Matplotlib doesn't come built in with Python's standard library, so you'll have to install it. The installation instructions are covered in Appendix A. Once you have it installed, start a Python shell. As explained in the installation instructions, you can either continue using IDLE shell or use Python's built-in shell.

Now we're ready to create our first graph. We'll start with a simple graph with just three points: (1, 2), (2, 4), and (3, 6). To create this graph, we'll first make two lists of numbers—one storing the values of the *x*-coordinates of these points and another storing the *y*-coordinates. The following two statements do exactly that, creating the two lists `x_numbers` and `y_numbers`:

```
>>> x_numbers = [1, 2, 3]
>>> y_numbers = [2, 4, 6]
```

From here, we can create the plot:

```
>>> from pylab import plot, show
>>> plot(x_numbers, y_numbers)
[<matplotlib.lines.Line2D object at 0x7f83ac60df10>]
```

In the first line, we import the `plot()` and `show()` functions from the `pylab` module, which is part of the matplotlib package. Next, we call the `plot()` function in the second line. The first argument to the `plot()` function is the list of numbers we want to plot on the *x*-axis, and the second argument is the corresponding list of numbers we want to plot on the *y*-axis. The `plot()` function returns an object—or more precisely, a list containing an object. This object contains the information about the graph that we asked Python to create. At this stage, you can add more information, such as a title, to the graph, or you can just display the graph as it is. For now we'll just display the graph.

The `plot()` function only creates the graph. To actually display it, we have to call the `show()` function:

```
>>> show()
```

You should see the graph in a matplotlib window as shown in Figure 2-3. (The display window may look different depending on your operating system, but the graph should be the same.)

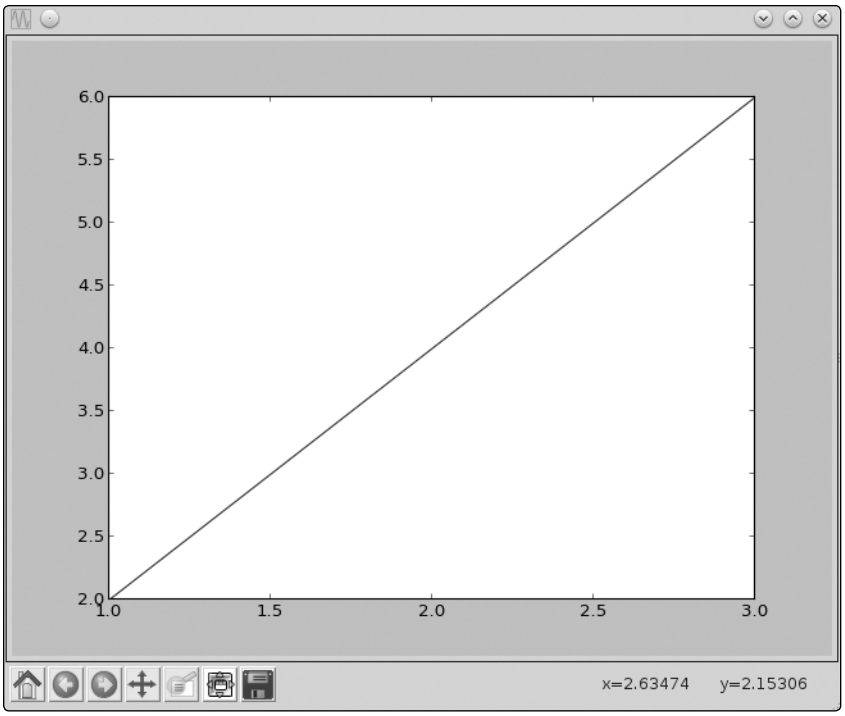

Figure 2-3: A graph showing a line passing through the points (1, 2), (2, 4), and (3, 6)

Notice that instead of starting from the origin (0, 0), the *x*-axis starts from the number 1 and the *y*-axis starts from the number 2. These are the lowest numbers from each of the two lists. Also, you can see increments marked on each of the axes (such as 2.5, 3.0, 3.5, etc., on the *y*-axis). In "Customizing Graphs" on page 41, we'll learn how to control those aspects of the graph, along with how to add axes labels and a graph title.

You'll notice in the interactive shell that you can't enter any further statements until you close the matplotlib window. Close the graph window so that you can continue programming.

Marking Points on Your Graph

If you want the graph to mark the points that you supplied for plotting, you can use an additional keyword argument while calling the `plot()` function:

```
>>> plot(x_numbers, y_numbers, marker='o')
```

By entering `marker='o'`, we tell Python to mark each point from our lists with a small dot that looks like an *o*. Once you enter `show()` again, you'll see that each point is marked with a dot (see Figure 2-4).

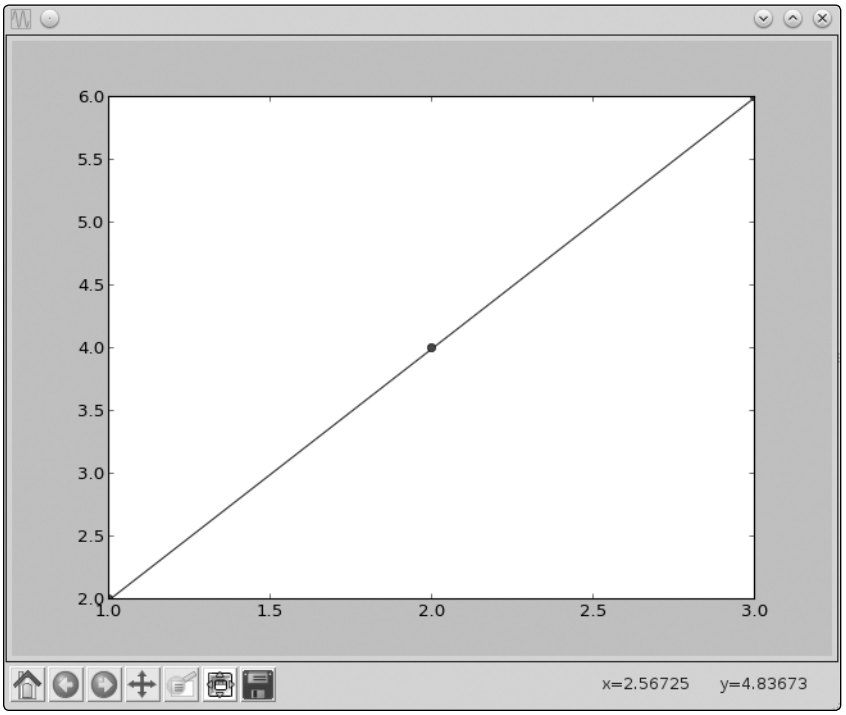

Figure 2-4: A graph showing a line passing through the points (1, 2), (2, 4), and (3, 6) with the points marked by a dot

The marker at (2, 4) is easily visible, while the others are hidden in the very corners of the graph. You can choose from several `marker` options, including `'o'`, `'*'`, `'x'`, and `'+'`. Using `marker=` includes a line connecting the points (this is the default). You can also make a graph that marks only the points that you specified, without any line connecting them, by omitting `marker=`:

```
>>> plot(x_numbers, y_numbers, 'o')
[<matplotlib.lines.Line2D object at 0x7f2549bc0bd0>]
```

Here, `'o'` indicates that each point should be marked with a dot, but there should be no line connecting the points. Call the function `show()` to display the graph, which should look like the one shown in Figure 2-5.

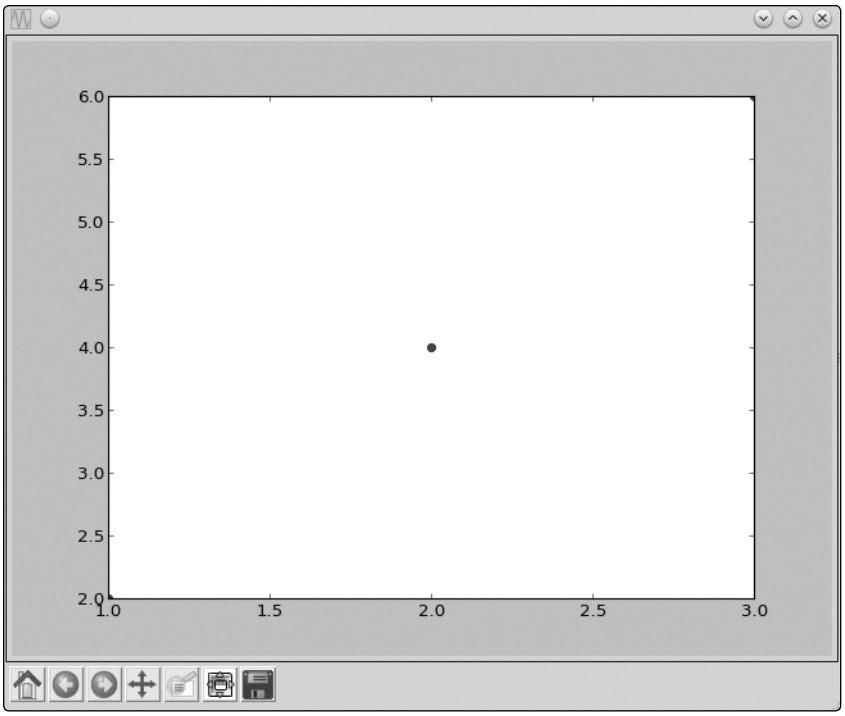

Figure 2-5: A graph showing the points (1, 2), (2, 4), and (3, 6)

As you can see, only the points are now shown on the graph, with no line connecting them. As in the previous graph, the first and the last points are barely visible, but we'll soon see how to change that.

Graphing the Average Annual Temperature in New York City

Let's take a look at a slightly larger set of data so we can explore more features of matplotlib. The average annual temperatures for New York City—measured at Central Park, specifically—during the years 2000 to 2012 are as follows: 53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, and 57.3 degrees Fahrenheit. Right now, that just looks like a random jumble of numbers, but we can plot this set of temperatures on a graph to make the rise and fall in the average temperature from year to year much clearer:

```
>>> nyc_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]
>>> plot(nyc_temp, marker='o')
[<matplotlib.lines.Line2D object at 0x7f2549d52f90>]
```

We store the average temperatures in a list, `nyc_temp`. Then, we call the function `plot()` passing only this list (and the marker string). When you use `plot()` on a single list, those numbers are automatically plotted on the *y*-axis. The corresponding values on the *x*-axis are filled in as the positions of each value in the list. That is, the first temperature value, 53.9, gets a corresponding *x*-axis value of 0 because it's in position 0 of the list (remember, the list position starts counting from 0, not 1). As a result, the numbers plotted on the *x*-axis are the integers from 0 to 12, which we can think of as corresponding to the 13 years for which we have temperature data.

Enter `show()` to display the graph, which is shown in Figure 2-6. The graph shows that the average temperature has risen and fallen from year to year. If you glance at the numbers we plotted, they really aren't very far apart from each other. However, the graph makes the variations seem rather dramatic. So, what's going on? The reason is that matplotlib chooses the range of the *y*-axis so that it's just enough to enclose the data supplied for plotting. So in this graph, the *y*-axis starts at 53.0 and its highest value is 57.5. This makes even small differences look magnified because the range of the *y*-axis is so small. We'll learn how to control the range of each axis in "Customizing Graphs" on page 41.

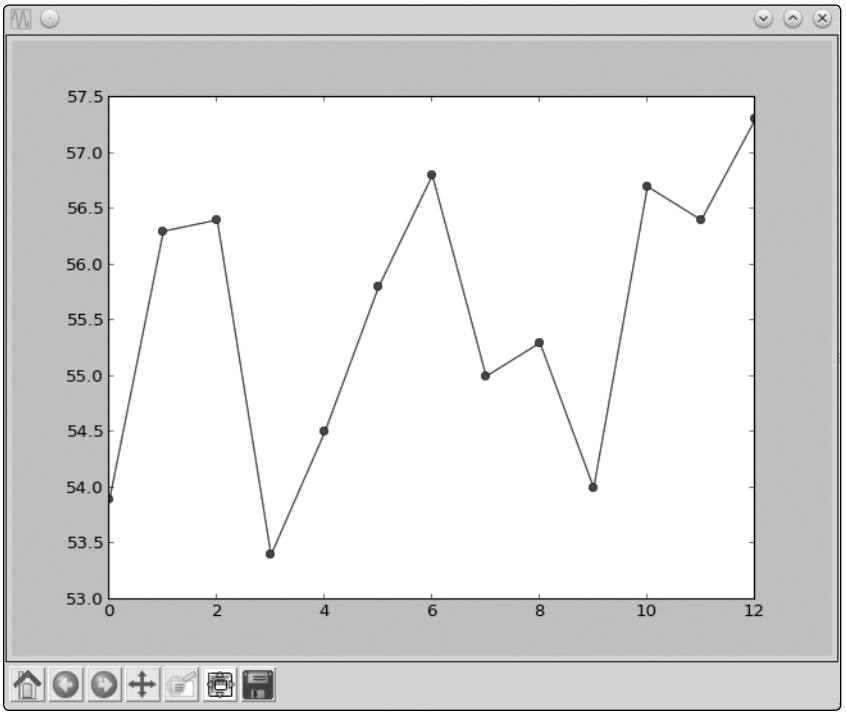

Figure 2-6: A graph showing the average annual temperature of New York City during the years 2000–2012

You can also see that numbers on the *y*-axis are floating point numbers (because that's what we asked to be plotted) and those on the *x*-axis are integers. Matplotlib can handle either.

Plotting the temperature without showing the corresponding years is a quick and easy way to visualize the variations between the years. If you were planning to present this graph to someone, however, you'd want to make it clearer by showing which year each temperature corresponds to. We can easily do this by creating another list with the years in it and then calling the `plot()` function:

```
>>> nyc_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]
>>> years = range(2000, 2013)
>>> plot(years, nyc_temp, marker='o')
[<matplotlib.lines.Line2D object at 0x7f2549a616d0>]
>>> show()
```

We use the `range()` function we learned about in Chapter 1 to specify the years 2000 to 2012. Now you'll see the years displayed on the *x*-axis (see Figure 2-7).

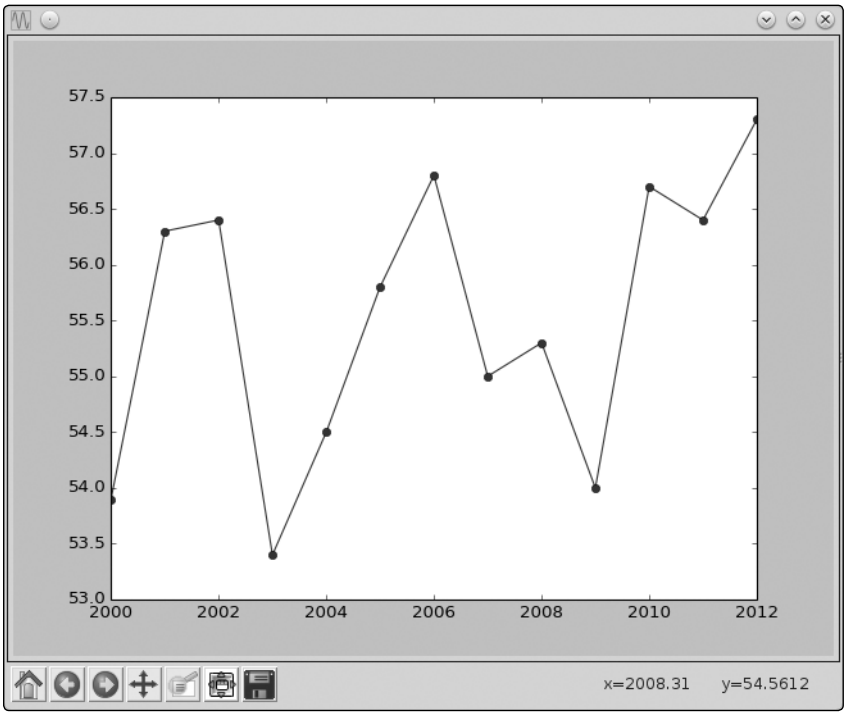

Figure 2-7: A graph showing the average annual temperature of New York City, displaying the years on the x-axis

Comparing the Monthly Temperature Trends of New York City

While still looking at New York City, let's see how the average monthly temperature has varied over the years. This will give us a chance to understand how to plot multiple lines on a single graph. We'll choose three years: 2000, 2006, and 2012. For each of these years, we'll plot the average temperature for all 12 months.

First, we need to create three lists to store the temperature (in Fahrenheit). Each list will consist of 12 numbers corresponding to the average temperature from January to December each year:

```
>>> nyc_temp_2000 = [31.3, 37.3, 47.2, 51.0, 63.5, 71.3, 72.3, 72.7, 66.0, 57.0, 45.3, 31.1]
>>> nyc_temp_2006 = [40.9, 35.7, 43.1, 55.7, 63.1, 71.0, 77.9, 75.8, 66.6, 56.2, 51.9, 43.6]
>>> nyc_temp_2012 = [37.3, 40.9, 50.9, 54.8, 65.1, 71.0, 78.8, 76.7, 68.8, 58.0, 43.9, 41.5]
```

The first list corresponds to the year 2000, and the next two lists correspond to the years 2006 and 2012, respectively. We could plot the three sets of data on three different graphs, but that wouldn't make it very easy to see how each year compares to the others. Try doing it!

The clearest way to compare all of these temperatures is to plot all three data sets on a *single* graph, like this:

```
>>> months = range(1, 13)
>>> plot(months, nyc_temp_2000, months, nyc_temp_2006, months, nyc_temp_2012)
[<matplotlib.lines.Line2D object at 0x7f2549c1f0d0>, <matplotlib.lines.Line2D
object at 0x7f2549a61150>, <matplotlib.lines.Line2D object at 0x7f2549c1b550>]
```

First, we create a list (`months`) where we store the numbers 1, 2, 3, and so on up to 12 using the `range()` function. Next, we call the `plot()` function with three pairs of lists. Each pair consists of a list of months to be plotted on the *x*-axis and a list of average monthly temperatures (for 2000, 2006, and 2012, respectively) to be plotted on the *y*-axis. So far, we've used `plot()` on only one pair of lists at a time, but you can actually enter multiple pairs of lists into the `plot()` function. With each list separated by a comma, the `plot()` function will automatically plot a different line for each pair.

The `plot()` function returns a list of three objects instead of one. Matplotlib considers the three curves as distinct from each other, and it knows to draw them on top of each other when you call `show()`. Let's call `show()` to display the graph, as shown in Figure 2-8.

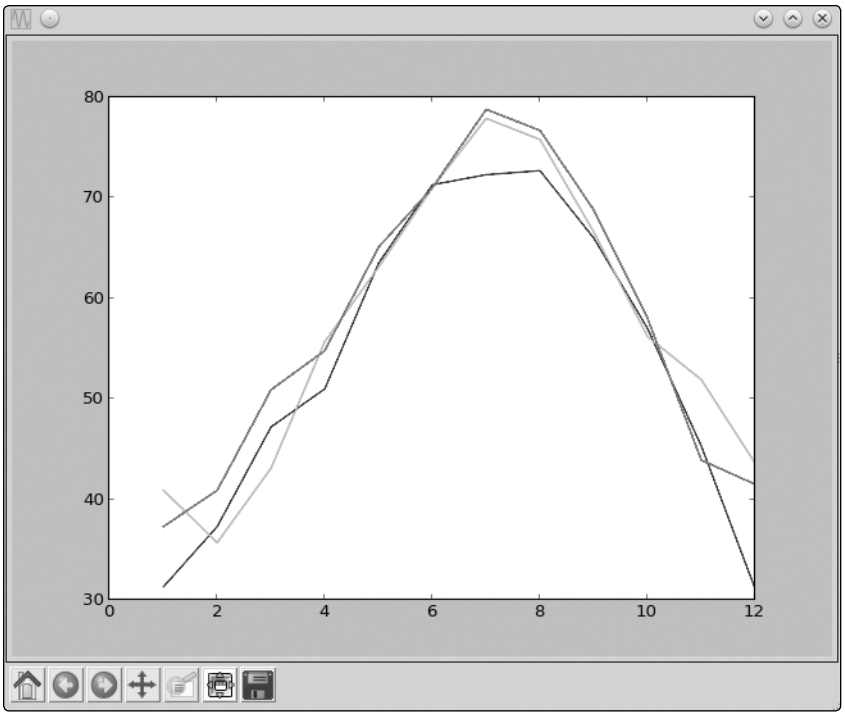

Figure 2-8: A graph showing the average monthly temperature of New York City during the years 2000, 2006, and 2012

Now we have three plots all on one graph. Python automatically chooses a different color for each line to indicate that the lines have been plotted from different data sets.

Instead of calling the plot function with all three pairs at once, we could also call the plot function three separate times, once for each pair:

```
>>> plot(months, nyc_temp_2000)
[<matplotlib.lines.Line2D object at 0x7f1e51351810>]
>>> plot(months, nyc_temp_2006)
[<matplotlib.lines.Line2D object at 0x7f1e5ae8e390>]
>>> plot(months, nyc_temp_2012)
[<matplotlib.lines.Line2D object at 0x7f1e5136ccd0>]
>>> show()
```

Matplotlib keeps track of what plots haven't been displayed yet. So as long as we wait to call `show()` until after we call `plot()` all three times, the plots will all get displayed on the same graph.

We have a problem, however, because we don't have any clue as to which color corresponds to which year. To fix this, we can use the function `legend()`, which lets us add a legend to the graph. A *legend* is a small display box that identifies what different parts of the graph mean. Here, we'll use a legend to indicate which year each colored line stands for. To add the legend, first call the `plot()` function as earlier:

```
>>> plot(months, nyc_temp_2000, months, nyc_temp_2006, months, nyc_temp_2012)
[<matplotlib.lines.Line2D object at 0x7f2549d6c410>, <matplotlib.lines.Line2D
object at 0x7f2549d6c9d0>, <matplotlib.lines.Line2D object at 0x7f2549a86850>]
```

Then, import the `legend()` function from the `pylab` module and call it as follows:

```
>>> from pylab import legend
>>> legend([2000, 2006, 2012])
<matplotlib.legend.Legend object at 0x7f2549d79410>
```

We call the `legend()` function with a list of the labels we want to use to identify each plot on the graph. These labels are entered in this order to match the order of the pairs of lists that were entered in the `plot()` function. That is, `2000` will be the label for the plot of the first pair we entered in the `plot()` function; `2006`, for the second pair; and `2012`, for the third. You can also specify a second argument to the function that will specify the position of the legend. By default, it's always positioned at the top right of the graph. However, you can specify a particular position, such as `'lower center'`, `'center left'`, and `'upper left'`. Or you can set the position to `'best'`, and the legend will be positioned so as not to interfere with the graph.

Finally, we call `show()` to display the graph:

```
>>> show()
```

As you can see in the graph (see Figure 2-9), there's now a legend box in the top-right corner. It tells us which line represents the average monthly temperature for the year 2000, which line represents the year 2006, and which line represents the year 2012.

Looking at the graph, you can conclude two interesting facts: the highest temperature for all three years was in and around July (corresponding to 7 on the *x*-axis), and it has been increasing from 2000 with a more dramatic rise between 2000 and 2006. Having all three lines plotted together in one graph makes it a lot easier to see these kinds of relationships. It's certainly clearer than just looking at a few long lists of numbers or even looking at three lines plotted on three separate graphs.

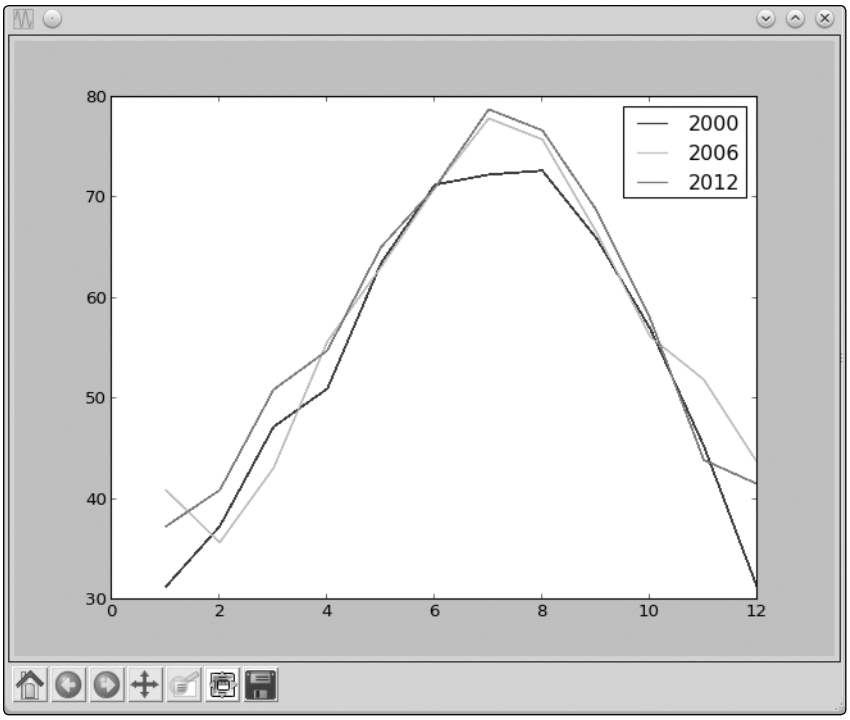

Figure 2-9: A graph showing the average monthly temperature of New York City, with a legend to show the year each color corresponds to

Customizing Graphs

We already learned about one way to customize a graph—by adding a legend. Now, we'll learn about other ways to customize a graph and to make it clearer by adding labels to the *x*- and *y*-axes, adding a title to the graph, and controlling the range and steps of the axes.

Adding a Title and Labels

We can add a title to our graph using the `title()` function and add labels for the *x*- and *y*-axes using the `xlabel()` and `ylabel()` functions. Let's re-create the last plot and add all this additional information:

```
>>> from pylab import plot, show, title, xlabel, ylabel, legend
>>> plot(months, nyc_temp_2000, months, nyc_temp_2006, months, nyc_temp_2012)
[<matplotlib.lines.Line2D object at 0x7f2549a9e210>, <matplotlib.lines.Line2D
object at 0x7f2549a4be90>, <matplotlib.lines.Line2D object at 0x7f2549a82090>]
>>> title('Average monthly temperature in NYC')
<matplotlib.text.Text object at 0x7f25499f7150>
>>> xlabel('Month')
<matplotlib.text.Text object at 0x7f2549d79210>
>>> ylabel('Temperature')
<matplotlib.text.Text object at 0x7f2549b8b2d0>
```

```
>>> legend([2000, 2006, 2012])
<matplotlib.legend.Legend object at 0x7f2549a82910>
```

All three functions—`title()`, `xlabel()`, and `ylabel()`—are called with the corresponding text that we want to appear on the graph entered as strings. Calling the `show()` function will display the graph with all this newly added information (see Figure 2-10).

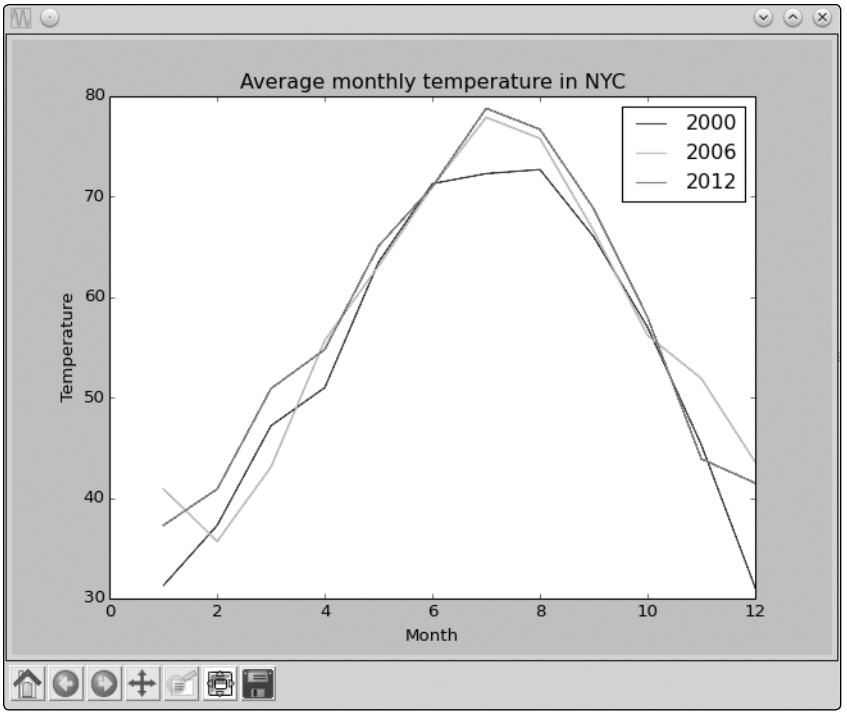

Figure 2-10: Axes labels and a title have been added to the graph.

With the three new pieces of information added, the graph is easier to understand.

Customizing the Axes

So far, we've allowed the numbers on both axes to be automatically determined by Python based on the data supplied to the `plot()` function. This may be fine for most cases, but sometimes this automatic range isn't the clearest way to present the data, as we saw in the graph where we plotted the average annual temperature of New York City (see Figure 2-7). There, even small changes in the temperature seemed large because the automatically chosen *y*-axis range was very narrow. We can adjust the range of the axes using the `axis()` function. This function can be used both to retrieve the current range and to set a new range for the axes.

Consider, once again, the average annual temperature of New York City during the years 2000 to 2012 and create a plot as we did earlier.

```
>>> nyc_temp = [53.9, 56.3, 56.4, 53.4, 54.5, 55.8, 56.8, 55.0, 55.3, 54.0, 56.7, 56.4, 57.3]
>>> plot(nyc_temp, marker='o')
[<matplotlib.lines.Line2D object at 0x7f3ae5b767d0>]
```

Now, import the `axis()` function and call it:

```
>>> from pylab import axis
>>> axis()
(0.0, 12.0, 53.0, 57.5)
```

The function returned a tuple with four numbers corresponding to the range for the *x*-axis (0.0, 12.0) and the *y*-axis (53.0, 57.5). These are the same range values from the graph that we made earlier. Now, let's change the *y*-axis to start from 0 instead of 53.0:

```
>>> axis(ymin=0)
(0.0, 12.0, 0, 57.5)
```

Calling the `axis()` function with the new starting value for the *y*-axis (specified by `ymin=0`) changes the range, and the returned tuple confirms it. If you display the graph by calling the `show()` function, the *y*-axis starts at 0, and the differences between the values of the consecutive years look less drastic (see Figure 2-11).

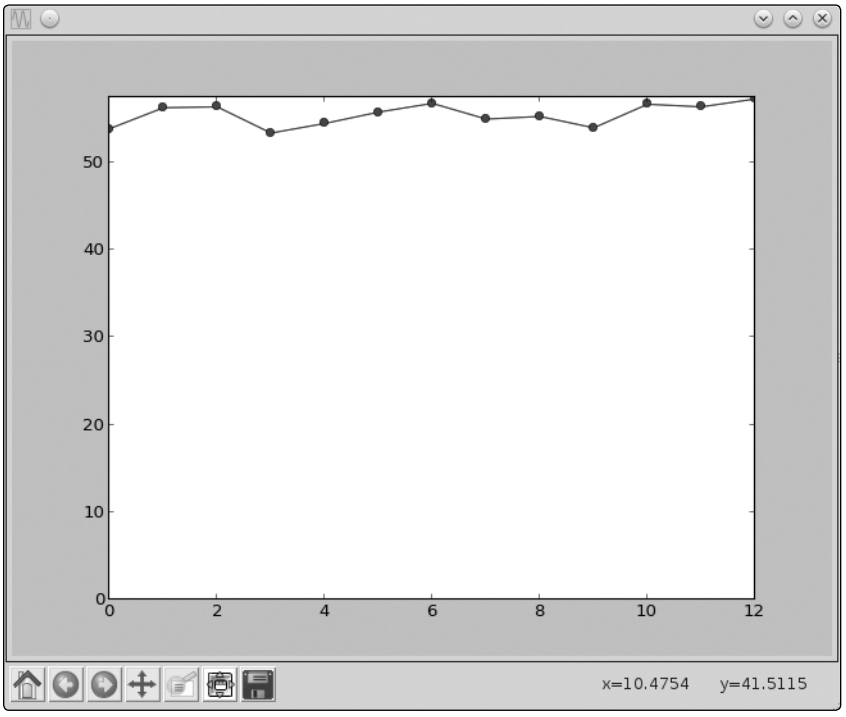

Figure 2-11: A graph showing the average annual temperature of New York City during the years 2000–2012. The y-axis has been customized to start from 0.

Similarly, you can use `xmin`, `xmax`, and `ymax` to set the minimum and maximum values for the *x*-axis and the maximum value for the *y*-axis, respectively. If you're changing all four values, you may find it easier to call the `axis()` function with all four range values entered as a list, such as `axis([0, 10, 0, 20])`. This would set the range of the *x*-axis to (0, 10) and that of the *y*-axis to (0, 20).

Plotting Using pyplot

The `pylab` module is useful for creating plots in an interactive shell, such as the IDLE shell, as we've been doing so far. However, when using matplotlib outside of the IDLE shell—for example, as part of a larger program—the `pyplot` module is more efficient. Don't worry—all the methods that you learned about when using `pylab` will work the same way with `pyplot`.

The following program recreates the first plot in this chapter using the `pyplot` module:

```
'''
Simple plot using pyplot
'''
❶ import matplotlib.pyplot

❷ def create_graph():
    x_numbers = [1, 2, 3]
    y_numbers = [2, 4, 6]

    matplotlib.pyplot.plot(x_numbers, y_numbers)
    matplotlib.pyplot.show()

if __name__ == '__main__':
    create_graph()
```

First, we import the `pyplot` module using the statement `import matplotlib.pyplot` ❶. This means that we're importing the entire `pyplot` module from the matplotlib package. To refer to any function or class definition defined in this module, you'll have to use the syntax `matplotlib.pyplot.`*`item`*, where *`item`* is the function or class you want to use.

This is different from importing a single function or class at a time, which is what we've been doing so far. For example, in the first chapter we imported the `Fraction` class as `from fractions import Fraction`. Importing an entire module is useful when you're going to use a number of functions from that module. Instead of importing them individually, you can just import the whole module at once and refer to different functions when you need them.

In the `create_graph()` function at ❷, we create the two lists of numbers that we want to plot on the graph and then pass the two lists to the `plot()` function, the same way we did before with `pylab`. This time, however, we call the function as `matplotlib.pyplot.plot()`, which means that we're calling the `plot()` function defined in the `pyplot` module of the matplotlib package. Then, we call the `show()` function to display the graph. The only difference

between the way you plot the numbers here compared to what we did earlier is the mechanism of calling the functions.

To save us some typing, we can import the `pyplot` module by entering `import matplotlib.pyplot as plt`. Then, we can refer to `pyplot` with the label `plt` in our programs, instead of having to always type `matplotlib.pyplot`:

```
'''
Simple plot using pyplot
'''
import matplotlib.pyplot as plt

def create_graph():
    x_numbers = [1, 2, 3]
    y_numbers = [2, 4, 6]
    plt.plot(x_numbers, y_numbers)
    plt.show()

if __name__ == '__main__':
    create_graph()
```

Now, we can call the functions by prefixing them with the shortened `plt` instead of `matplotlib.pyplot`.

Going ahead, for the rest of this chapter and this book, we'll use `pylab` in the interactive shell and `pyplot` otherwise.

Saving the Plots

If you need to save your graphs, you can do so using the `savefig()` function. This function saves the graph as an image file, which you can use in reports or presentations. You can choose among several image formats, including PNG, PDF, and SVG.

Here's an example:

```
>>> from pylab import plot, savefig
>>> x = [1, 2, 3]
>>> y = [2, 4, 6]
>>> plot(x, y)
>>> savefig('mygraph.png')
```

This program will save the graph to an image file, *mygraph.png*, in your current directory. On Microsoft Windows, this is usually *C:\Python33* (where you installed Python). On Linux, the current directory is usually your home directory *(/home/<username>)*, where *<username>* is the user you're logged in as. On a Mac, IDLE saves files to *~/Documents* by default. If you wanted to save it in a different directory, specify the complete pathname. For example, to save the image under *C:* on Windows as *mygraph.png*, you'd call the `savefig()` function as follows:

```
>>> savefig('C:\mygraph.png')
```

If you open the image in an image-viewing program, you'll see the same graph you'd see by calling the `show()` function. (You'll notice that the image file contains only the graph—not the entire window that pops up with the `show()` function). To specify a different image format, simply name the file with the appropriate extension. For example, `mygraph.svg` will create an SVG image file.

Another way to save a figure is to use the Save button in the window that pops up when you call `show()`.

Plotting with Formulas

Until now, we've been plotting points on our graphs based on observed scientific measurements. In those graphs, we already had all our values for x and y laid out. For example, recorded temperatures and dates were already available to us at the time we wanted to create the New York City graph, showing how the temperature varied over months or years. In this section, we're going to create graphs from mathematical formulas.

Newton's Law of Universal Gravitation

According to Newton's law of universal gravitation, a body of mass m_1 attracts another body of mass m_2 with an amount of force F according to the formula

$$F = \frac{Gm_1m_2}{r^2},$$

where r is the distance between the two bodies and G is the gravitational constant. We want to see what happens to the force as the distance between the two bodies increases.

Let's take the masses of two bodies: the mass of the first body (m_1) is 0.5 kg, and the mass of the second body (m_2) is 1.5 kg. The value of the gravitational constant is 6.674×10^{-11} N m^2 kg^{-2}. Now we're ready to calculate the gravitational force between these two bodies at 19 different distances: 100 m, 150 m, 200 m, 250 m, 300 m, and so on up through 1000 m. The following program performs these calculations and also draws the graph:

```
'''
The relationship between gravitational force and
distance between two bodies
'''

import matplotlib.pyplot as plt

# Draw the graph
def draw_graph(x, y):
    plt.plot(x, y, marker='o')
    plt.xlabel('Distance in meters')
```

```
    plt.ylabel('Gravitational force in newtons')
    plt.title('Gravitational force and distance')
    plt.show()

def generate_F_r():
    # Generate values for r
❶   r = range(100, 1001, 50)
    # Empty list to store the calculated values of F
    F = []

    # Constant, G
    G = 6.674*(10**-11)
    # Two masses
    m1 = 0.5
    m2 = 1.5

    # Calculate force and add it to the list, F
❷   for dist in r:
        force = G*(m1*m2)/(dist**2)
        F.append(force)

    # Call the draw_graph function
❸   draw_graph(r, F)

if __name__=='__main__':
    generate_F_r()
```

The `generate_F_r()` function does most of the work in the program above. At ❶, we use the `range()` function to create a list labeled `r` with different values for distance, using a step value of 50. The final value is specified as 1001 because we want 1000 to be included as well. We then create an empty list (`F`), where we'll store the corresponding gravitational force at each of these distances. Next, we create labels referring to the gravitational constant (`G`) and the two masses (`m1` and `m2`). Using a `for` loop ❷, we then calculate the force at each of the values in the list of distances (`r`). We use a label (`force`) to refer to the force calculated and to append it to the list (`F`). Finally, we call the function `draw_graph()` at ❸ with the list of distances and the list of the calculated forces. The *x*-axis of the graph displays the force, and the *y*-axis displays the distance. The graph is shown in Figure 2-12.

As the distance (`r`) increases, the gravitational force decreases. With this kind of relationship, we say that the gravitational force is *inversely proportional* to the distance between the two bodies. Also, note that when the value of one of the two variables changes, the other variable won't necessarily change by the same proportion. We refer to this as a *nonlinear relationship*. As a result, we end up with a curved line on the graph instead of a straight one.

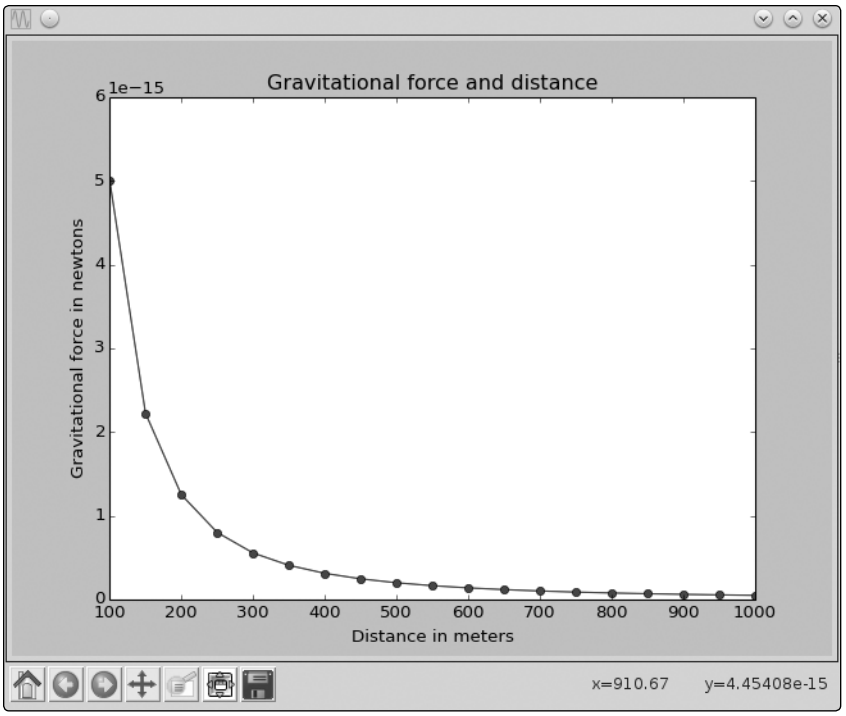

Figure 2-12: Visualization of the relationship between the gravitational force and the squared distance

Projectile Motion

Now, let's graph something you'll be familiar with from everyday life. If you throw a ball across a field, it follows a trajectory like the one shown in Figure 2-13.

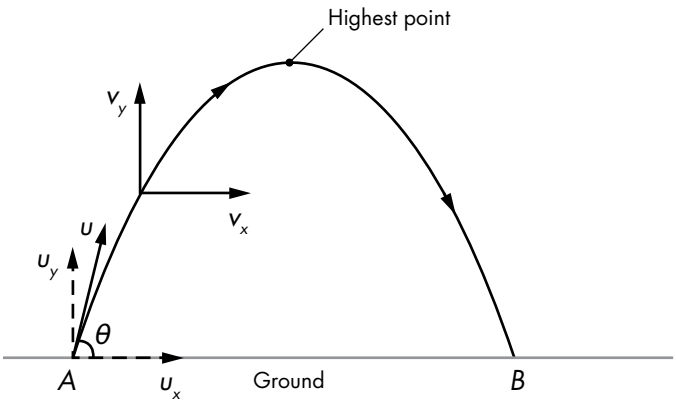

Figure 2-13: Motion of a ball that's thrown at point A*—at an angle (*θ*) with a velocity (*u*)—and that hits the ground at point* B

In the figure, the ball is thrown from point *A* and lands at point *B*. This type of motion is referred to as *projectile* motion. Our aim here is to use the equations of projectile motion to graph the trajectory of a body, showing the position of the ball starting from the point it's thrown until it hits the ground again.

When you throw the ball, it has an initial velocity and the direction of that velocity creates a certain angle with the ground. Let's call the initial velocity u and the angle that it makes with the ground θ (theta), as shown in Figure 2-13. The ball has two velocity components: one along the x direction, calculated by $u_x = u\cos\theta$, and the other along the y direction, where $u_y = u\sin\theta$.

As the ball moves, its velocity changes, and we will represent that changed velocity using v: the horizontal component is v_x and the vertical component is v_y. For simplicity, assume the horizontal component (v_x) doesn't change during the motion of the body, whereas the vertical component (v_y) decreases because of the force of gravity according to the equation $v_y = u_y - gt$. In this equation, g is the gravitational acceleration and t is the time at which the velocity is measured. Because $u_y = u\sin\theta$, we can substitute to get

$$v_y = u\sin\theta - gt.$$

Because the horizontal component of the velocity remains constant, the horizontal distance traveled (S_x) is given by $S_x = u(\cos\theta)t$. The vertical component of the velocity changes, though, and the vertical distance traveled is given by the formula

$$S_y = u(\sin\theta)t - \frac{1}{2}gt^2.$$

In other words, S_x and S_y give us the x- and y-coordinates of the ball at any given point in time during its flight. We'll use these equations when we write a program to draw the trajectory. As we use these equations, time (t) will be expressed in seconds, the velocity will be expressed in m/s, the angle of projection (θ) will be expressed in degrees, and the gravitational acceleration (g) will be expressed in m/s^2.

Before we write our program, however, we'll need to find out how long the ball will be in flight before it hits the ground so that we know when our program should stop plotting the trajectory of the ball. To do so, we'll first find how long the ball takes to reach its highest point. The ball reaches its highest point when the vertical component of the velocity (v_y) is 0, which is when $v_y = u\sin\theta - gt = 0$. So we're looking for the value t using the formula

$$t = \frac{u\sin\theta}{g}.$$

We'll call this time `t_peak`. After it reaches its highest point, the ball will hit the ground after being airborne for another `t_peak` seconds, so the total time of flight (`t_flight`) of the ball is

$$t_{\text{flight}} = 2t_{\text{peak}} = 2\frac{u\sin\theta}{g}.$$

Let's take a ball that's thrown with an initial velocity (u) of 5 m/s at an angle (θ) of 45 degrees. To calculate the total time of flight, we substitute $u = 5$, $\theta = 45$, and $g = 9.8$ into the equation we saw above:

$$t_{\text{flight}} = 2\frac{5\sin 45}{9.8}.$$

In this case, the time of flight for the ball turns out to be 0.72154 seconds (rounded to five decimal places). The ball will be in air for this period of time, so to draw the trajectory, we'll calculate its x- and y-coordinates at regular intervals during this time period. How often should we calculate the coordinates? Ideally, as frequently as possible. In this chapter, we'll calculate the coordinates every 0.001 seconds.

Generating Equally Spaced Floating Point Numbers

We've used the `range()` function to generate equally spaced integers—that is, if we wanted a list of integers between 1 and 10 with each integer separated by 1, we would use `range(1, 10)`. If we wanted a different step value, we could specify that to the range function as the third argument. Unfortunately, there's no such built-in function for floating point numbers. So, for example, there's no function that would allow us to create a list of the numbers from 0 to 0.72 with two consecutive numbers separated by 0.001. We can use a `while` loop as follows to create our own function for this:

```
'''
Generate equally spaced floating point
numbers between two given values
'''

def frange(start, final, increment):

    numbers = []
❶   while start < final:
❷       numbers.append(start)
        start = start + increment

    return numbers
```

We've defined a function `frange()` ("floating point" range) that receives three parameters: `start` and `final` refer to the starting and the final points of the range of numbers, and `increment` refers to the difference between two consecutive numbers. We initialize a `while` loop at ❶, which continues execution as long as the number referred to by `start` is less than the value for `final`. We store the number pointed to by `start` in the list `numbers` ❷ and then add the value we entered as an `increment` during every iteration of the loop. Finally, we return the list `numbers`.

We'll use this function to generate equally spaced time instants in the trajectory-drawing program described next.

Drawing the Trajectory

The following program draws the trajectory of a ball thrown with a certain velocity and angle—both of which are supplied as input to the program:

```
'''
Draw the trajectory of a body in projectile motion
'''

from matplotlib import pyplot as plt
import math

def draw_graph(x, y):
    plt.plot(x, y)
    plt.xlabel('x-coordinate')
    plt.ylabel('y-coordinate')
    plt.title('Projectile motion of a ball')


def frange(start, final, interval):

    numbers = []
    while start < final:
        numbers.append(start)
        start = start + interval

    return numbers

def draw_trajectory(u, theta):

❶     theta = math.radians(theta)
      g = 9.8

      # Time of flight
❷     t_flight = 2*u*math.sin(theta)/g
      # Find time intervals
      intervals = frange(0, t_flight, 0.001)
```

```
    # List of x and y coordinates
    x = []
    y = []
❸   for t in intervals:
        x.append(u*math.cos(theta)*t)
        y.append(u*math.sin(theta)*t - 0.5*g*t*t)

    draw_graph(x, y)

if __name__ == '__main__':
❹   try:
        u = float(input('Enter the initial velocity (m/s): '))
        theta = float(input('Enter the angle of projection (degrees): '))
    except ValueError:
        print('You entered an invalid input')
    else:
        draw_trajectory(u, theta)
        plt.show()
```

In this program, we use the functions `radians()`, `cos()`, and `sin()` defined in the standard library's `math` module, so we import that module at the beginning. The `draw_trajectory()` function accepts two arguments, `u` and `theta`, corresponding to the velocity and the angle at which the ball is thrown. The `math` module's sine and the cosine functions expect the angle to be supplied in radians, so at ❶, we convert the angle (`theta`) from degrees to radians using the `math.radians()` function. Next, we create a label (`g`) to refer to the value of acceleration due to gravity, 9.8 m/s^2. At ❷, we calculate the time of flight and then call the `frange()` function with the values for `start`, `final`, and `increment` set to 0, `t_flight`, and 0.001, respectively. We then calculate the *x*- and *y*-coordinates for the trajectory at each of the time instants and store them in two separate lists, `x` and `y` ❸. To calculate these coordinates, we use the formulas for the distances S_x and S_y that we discussed earlier.

Finally, we call the `draw_graph()` function with the *x*- and *y*-coordinates to draw the trajectory. Note that the `draw_graph()` function doesn't call the `show()` function (we'll see why in the next program). We use a `try...except` block ❹ to report an error message in case the user enters an invalid input. Valid input for this program is any integer or floating point number. When you run the program, it asks for these values as input and then draws the trajectory (see Figure 2-14):

```
Enter the initial velocity (m/s): 25
Enter the angle of projection (degrees): 60
```

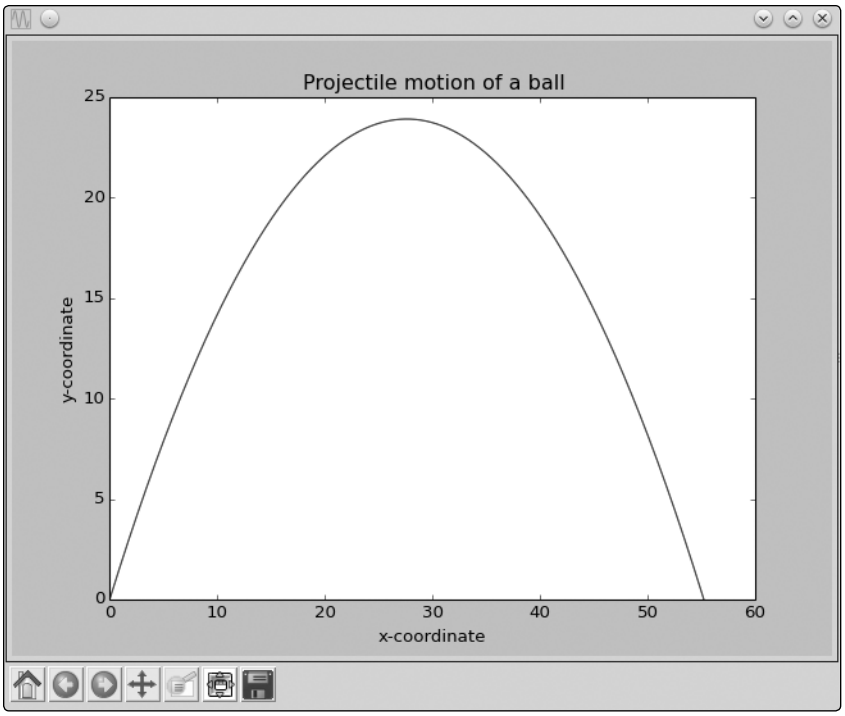

Figure 2-14: The trajectory of a ball when thrown with a velocity of 25 m/s at an angle of 60 degrees

Comparing the Trajectory at Different Initial Velocities

The previous program allows you to perform interesting experiments. For example, what will the trajectory look like for three balls thrown at different velocities but with the same initial angle? To graph three trajectories at once, we can replace the `main` code block from our previous program with the following:

```
if __name__ == '__main__':

    # List of three different initial velocities
❶   u_list = [20, 40, 60]
    theta = 45
    for u in u_list:
        draw_trajectory(u, theta)

    # Add a legend and show the graph
❷   plt.legend(['20', '40', '60'])
    plt.show()
```

Here, instead of asking the program's user to enter the velocity and the angle of projection, we create a list (`u_list`) with the velocities 20, 40, and 60 at ❶ and set the angle of projection as 45 degrees (using the label `theta`). We then call the `draw_trajectory()` function with each of the three values in `u_list` using the same value for `theta`, which calculates the list of *x*- and *y*-coordinates and calls the `draw_graph()` function. When we call the `show()` function, all three plots are displayed on the same graph. Because we now have a graph with multiple plots, we add a legend to the graph at ❷ before calling `show()` to display the velocity for each line. When you run the above program, you'll see the graph shown in Figure 2-15.

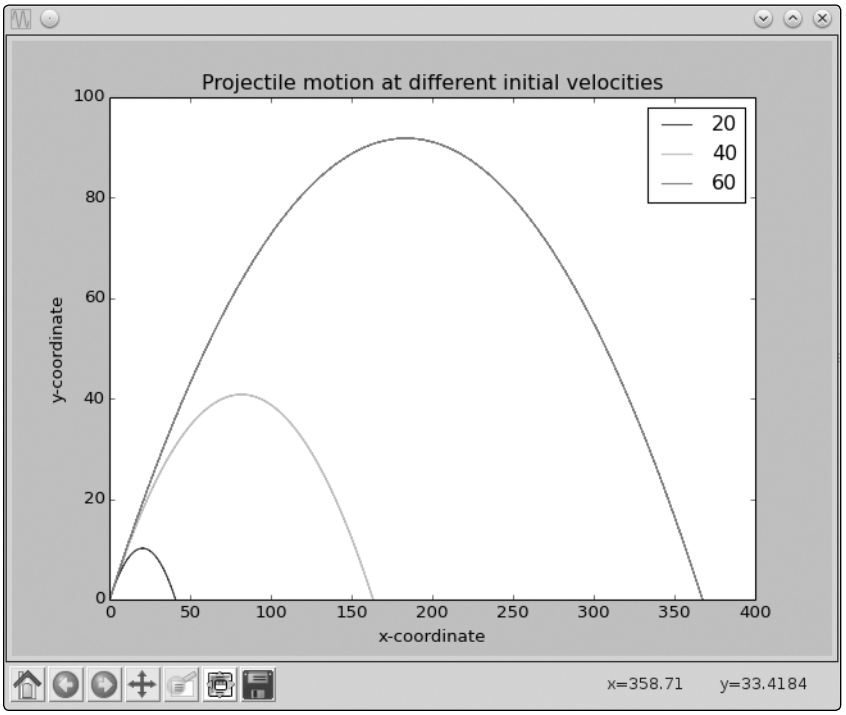

Figure 2-15: The trajectory of a ball thrown at a 60-degree angle, with a velocity of 20, 40, and 60 m/s

What You Learned

In this chapter, you learned the basics of creating graphs with matplotlib. You saw how to plot a single set of values, how to create multiple plots on the same graph, and how to label various parts of a graph to make it more informative. You used graphs to analyze the temperature variation of a city, study Newton's law of universal gravitation, and study the projectile motion of a body. In the next chapter, you'll use Python to start exploring statistics, and you'll see how drawing a graph can help make the relationships among sets of numbers easier to understand.

Programming Challenges

Here are a few challenges that build on what you've learned in this chapter. You can find sample solutions at *http://www.nostarch.com/doingmathwithpython/*.

#1: How Does the Temperature Vary During the Day?

If you enter a search term like "New York weather" in Google's search engine, you'll see, among other things, a graph showing the temperature at different times of the present day. Your task here is to re-create such a graph.

Using a city of your choice, find the temperature at different points of the day. Use the data to create two lists in your program and to create a graph with the time of day on the x-axis and the corresponding temperature on the y-axis. The graph should tell you how the temperature varies with the time of day. Try a different city and see how the two cities compare by plotting both lines on the same graph.

The time of day may be indicated by strings such as `'10:11 AM'` or `'09:21 PM'`.

#2: Exploring a Quadratic Function Visually

In Chapter 1, you learned how to find the roots of a quadratic equation, such as $x^2 + 2x + 1 = 0$. We can turn this equation into a function by writing it as $y = x^2 + 2x + 1$. For any value of x, the quadratic function produces *some* value for y. For example, when $x = 1$, $y = 4$. Here's a program that calculates the value of y for six different values of x:

```
'''
Quadratic function calculator
'''

# Assume values of x
❶ x_values = [-1, 1, 2, 3, 4, 5]
❷ for x in x_values:
    # Calculate the value of the quadratic function
    y = x**2 + 2*x + 1
    print('x={0} y={1}'.format(x, y))
```

At ❶, we create a list with six different values for `x`. The `for` loop starting at ❷ calculates the value of the function above for each of these values and uses the label `y` to refer to the list of results. Next, we print the value of `x` and the corresponding value of `y`. When you run the program, you should see the following output:

```
x=-1 y=0
x=1 y=4
x=2 y=9
```

```
x=3 y=16
x=4 y=25
x=5 y=36
```

Notice that the first line of the output is a root of the quadratic equation because it's a value for x that makes the function equal to 0.

Your programming challenge is to enhance this program to create a graph of the function. Try using at least 10 values for x instead of the 6 above. Calculate the corresponding y values using the function and then create a graph using these two sets of values.

Once you've created the graph, spend some time analyzing how the value of *y* varies with respect to *x*. Is the variation linear or nonlinear?

#3: Enhanced Projectile Trajectory Comparison Program

Your challenge here is to enhance the trajectory comparison program in a few ways. First, your program should print the time of flight, maximum horizontal distance, and maximum vertical distance traveled for each of the velocity and angle of projection combinations.

The other enhancement is to make the program work with any number of initial velocity and angle of projection values, supplied by the user. For example, here's how the program should ask the user for the inputs:

```
How many trajectories? 3
Enter the initial velocity for trajectory 1 (m/s): 45
Enter the angle of projection for trajectory 1 (degrees): 45
Enter the initial velocity for trajectory 2 (m/s): 60
Enter the angle of projection for trajectory 2 (degrees): 45
Enter the initial velocity for trajectory(m/s) 3: 45
Enter the angle of projection for trajectory(degrees) 3: 90
```

Your program should also ensure that erroneous input is properly handled using a `try...except` block, just as in the original program.

#4: Visualizing Your Expenses

I always find myself asking at the end of the month, "Where did all that money go?" I'm sure this isn't a problem I alone face.

For this challenge, you'll write a program that creates a bar chart for easy comparison of weekly expenditures. The program should first ask for the number of categories for the expenditures and the weekly total expenditure in each category, and then it should create the bar chart showing these expenditures.

Here's a sample run of how the program should work:

```
Enter the number of categories: 4
Enter category: Food
Expenditure: 70
```

```
Enter category: Transportation
Expenditure: 35
Enter category: Entertainment
Expenditure: 30
Enter category: Phone/Internet
Expenditure: 30
```

Figure 2-16 shows the bar chart that will be created to compare the expenditures. If you save the bar chart for every week, at the end of the month, you'll be able to see how the expenditures varied between the weeks for different categories.

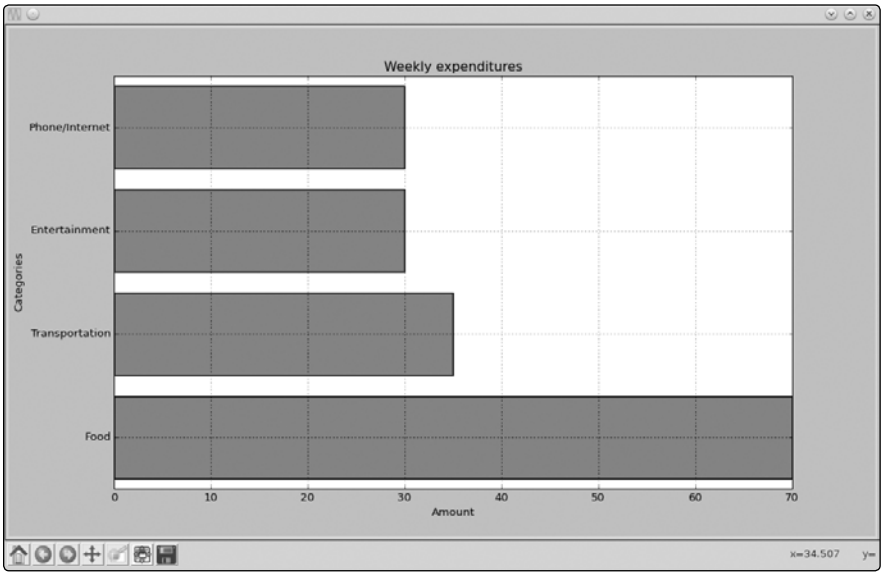

Figure 2-16: A bar chart showing the expenditures per category during the week

We haven't discussed creating a bar chart using matplotlib, so let's try an example.

A bar chart can be created using matplotlib's `barh()` function, which is also defined in the `pyplot` module. Figure 2-17 shows a bar chart that illustrates the number of steps I walked during the past week. The days of the week—Sunday, Monday, Tuesday, and so forth—are referred to as the *labels*. Each horizontal bar starts from the *y*-axis, and we have to specify the *y*-coordinate of the *center* of this position for each of the bars. The length of each bar corresponds to the number of steps specified.

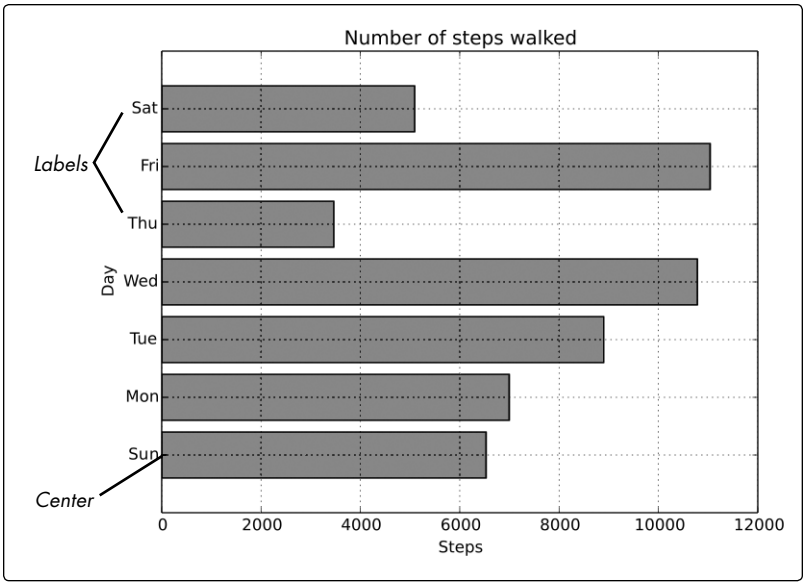

Figure 2-17: A bar chart showing the number of steps walked during a week

The following program creates the bar chart:

```
'''
Example of drawing a horizontal bar chart
'''
import matplotlib.pyplot as plt
def create_bar_chart(data, labels):
    # Number of bars
    num_bars = len(data)
    # This list is the point on the y-axis where each
    # Bar is centered. Here it will be [1, 2, 3...]
❶   positions = range(1, num_bars+1)
❷   plt.barh(positions, data, align='center')
    # Set the label of each bar
    plt.yticks(positions, labels)
    plt.xlabel('Steps')
    plt.ylabel('Day')
    plt.title('Number of steps walked')
    # Turns on the grid which may assist in visual estimation
    plt.grid(
    plt.show()

if __name__ == '__main__':
    # Number of steps I walked during the past week
    steps = [6534, 7000, 8900, 10786, 3467, 11045, 5095]
    # Corresponding days
    labels = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
    create_bar_chart(steps, labels)
```

The `create_bar_chart()` function accepts two parameters—`data`, which is a list of numbers we want to represent using the bars and labels, and the corresponding `labels` list. The center of each bar has to be specified, and I've arbitrarily chosen the centers as 1, 2, 3, 4, and so on using the help of the `range()` function at ❶.

We then call the `barh()` function, passing `positions` and `data` as the first two arguments and then the keyword argument, `align='center'`, at ❷. The keyword argument specifies that the bars are centered at the positions on the *y*-axis specified by the list. We then set the labels for each bar, the axis labels, and the title using the `yticks()` function. We also call the `grid()` function to turn on the grid, which may be useful for a visual estimation of the number of steps. Finally, we call the `show()` function.

#5: Exploring the Relationship Between the Fibonacci Sequence and the Golden Ratio

The Fibonacci sequence (1, 1, 2, 3, 5, . . .) is the series of numbers where the ith number in the series is the sum of the two previous numbers—that is, the numbers in the positions $(i - 2)$ and $(i - 1)$. The successive numbers in this series display an interesting relationship. As you increase the number of terms in the series, the ratios of consecutive pairs of numbers are nearly equal to each other. This value approaches a special number referred to as the *golden ratio*. Numerically, the golden ratio is the number 1.618033988 . . . , and it's been the subject of extensive study in music, architecture, and nature. For this challenge, write a program that will plot on a graph the ratio between consecutive Fibonacci numbers for, say, 100 numbers, which will demonstrate that the values approach the golden ratio.

You may find the following function, which returns a list of the first n Fibonacci numbers, useful in implementing your solution:

```
def fibo(n):
    if n == 1:
        return [1]
    if n == 2:
        return [1, 1]
    # n > 2
    a = 1
    b = 1
    # First two members of the series
    series = [a, b]
    for i in range(n):
        c = a + b
        series.append(c)
        a = b
        b = c

    return series
```

The output of your solution should be a graph, as shown in Figure 2-18.

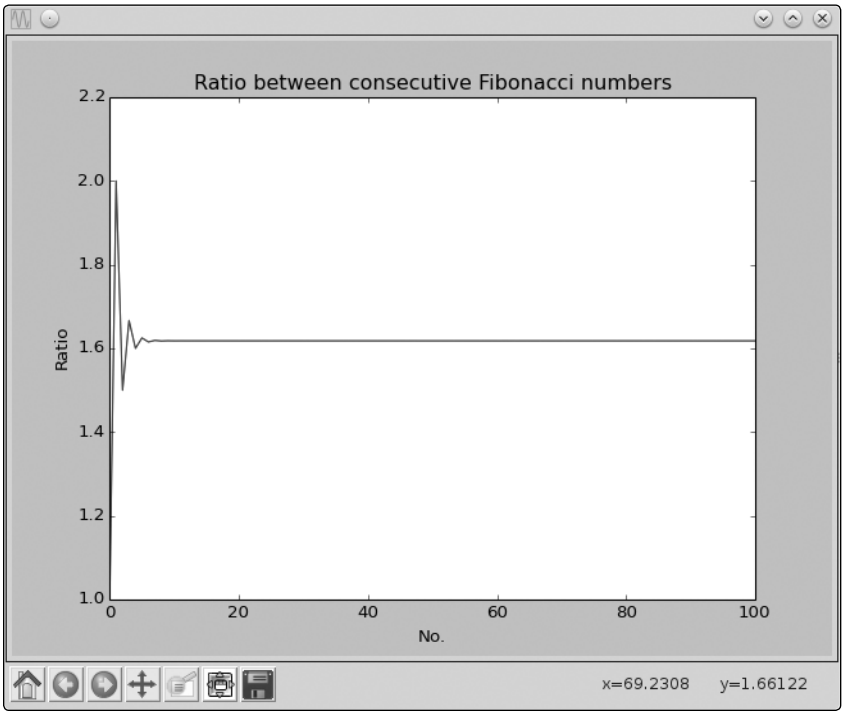

Figure 2-18: The ratio between the consecutive Fibonacci numbers approaches the golden ratio.